Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.A. Bergstra, J. Heering, P. Klint

ASF-An algebraic specification formalism

Computer Science/Department of Software Technology Report CS-R8705 January

69D21, 69F32

ASF — AN ALGEBRAIC SPECIFICATION FORMALISM

J.A. Bergstra
Department of Computer Science, University of Amsterdam
Department of Philosophy, University of Utrecht

J. Heering
Department of Software Technology, Centre for Mathematics and Computer Science

P. Klint
Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, University of Amsterdam

The algebraic specification formalism ASF supports modularized (first-order) equational specifications. Among its features are (a) import and parameterization of modules; (b) hidden (auxiliary) sorts and functions; (c) positive conditional equations; (d) overloaded functions; (e) infix operators. Most of the context-dependent errors in ASF specifications are violations of the so-called *origin rule*. Besides catching errors, this rule enforces a certain modularization of ASF specifications. The meaning of the modularization constructs of ASF is defined by means of a syntactical normalization procedure. Numerous examples of both correct and incorrect ASF specifications are given in an appendix.

Key Words & Phrases: Software Engineering, Specification Languages, Algebraic Specification Formalism, Modularization, Origin Rule, Module Expression, Normalization of Module Expressions.

1986 CR Categories: D.2.1 [**Software Engineering**]: Requirements, Specifications; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - *Algebraic Approaches to Semantics.*

1980 Mathematics Subject Classification: 68B99 [**Software**].

Note: Partial support received from the European Communities under ESPRIT projects 348 (Generation of Interactive Programming Environments — GIPE) and 432 (An Integrated Formal Approach to Industrial Software Development — METEOR), and from the Netherlands Organization for the Advancement of Pure Research (ZWO) (Compilation of Algebraic Specifications).

Report CS-R8705
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

TABLE OF CONTENTS

1. INTRODUCTION

In this report we give an informal description of the algebraic specification formalism ASF which is a minor revision of the formalism introduced in [BHK85]. The latter (locally known as the 'PICO-formalism') has been in use over the last two years for the specification of a variety of problems such as the dynamic semantics of a language with goto-statements [vD86], the static semantics of POOL [WAL86], the type-checking of a subset of ML [HEN87], and process algebra [MAU87].

The purpose of this report is to clarify some issues in the PICO-formalism and also to make several small improvements. Section 2 gives an informal definition of ASF. In section 3 we give some examples of ASF specifications and introduce 'structure diagrams' (a graphical representation of the modular structure of ASF specifications). The semantics of ASF specifications is described in section 4. The differences between the PICO-formalism and ASF as well as known shortcomings of ASF are listed in section 5. Finally, appendix I presents a set of examples of both correct and incorrect ASF specifications. These examples are part of a 'validation suite' for static consistency checkers for ASF.

2. INFORMAL DEFINITION OF ASF

A (many-sorted) *signature* is a set of declarations of *sorts* and *functions* over these sorts. A signature defines a language of strongly typed *terms* (*expressions*). A basic ASF *module* consists of a signature, a set of variable declarations, and a set of *positive conditional equations* in the language defined by the signature and the variable declarations. ASF modules may be *parameterized. Parameter binding* and *importing* modules in other ones are the two ways in which modules can be combined in ASF.

ASF *specifications* are sequences of modules. A module can be *normalized* in the context of a specification to which it belongs by eliminating all imports and binding as many parameters as possible. Normalization is a textual operation. The semantics of a module is the *initial algebra* [EM85, MG85] of its normal form, provided the latter does not have any remaining unbound parameters.

2.1. Syntax of ASF

In this section we give a context-free grammar for ASF. The following notational abbreviations are used in this definition:

- `[ <N> ]` denotes an optional occurrence of `<N>`.
- `<N>*` and `<N>+` denote, respectively, zero or more, and one or more occurrences of `<N>`.
- `{ <N> t }*` and `{ <N> t }+` denote, respectively, zero or more, and one or more occurrences of `<N>` separated by terminal symbol `t`.

ASF has the following grammar:

```
<specification>     ::= <module>+ .
<module>            ::= "module" <module-ident>
                        "begin"
                            [ <parameters> ]
                            [ <exports> ]
                            [ <imports> ]
                            [ <sorts> ]
                            [ <functions> ]
                            [ <variables> ]
                            [ <equations> ]
                        "end"  <module-ident> .
<module-ident>      ::= <ident> .
<parameters>        ::= "parameters" { <parameter>  ","}+ .
<parameter>         ::= <parameter-ident>
                        "begin"
                            [ <sorts> ]
                            [ <functions> ]
                        "end" <parameter-ident> .
```

```
<parameter-ident>    ::= <ident> .
<exports>            ::= "exports"
                         "begin"
                             [ <sorts> ]
                             [ <functions> ]
                         "end" .
<imports>            ::= "imports" { <module-expression> "," }+ .
<module-expression> ::= <module-ident> [ "{" <modifier> "}" ] .
<modifier>           ::= <renamed> [ <bound> ] | <bound> [ <renamed> ] .
<renamed>            ::= "renamed" "by" <renamings> .
<renamings>          ::= "[" { <renaming> "," }+ "]" .
<renaming>           ::= <sort> "->" <sort> |
                         <fun-or-operator-ident> "->" <fun-or-operator-ident> .
<fun-or-operator-ident>
                     ::= <fun-ident> | "_" <operator> "_" | <operator> "_" .
<bound>              ::= ( <parameter-ident> "bound" "by" <renamings>
                                                  "to"  <module-ident> )+ .
<sorts>              ::= "sorts" <sort-list> .
<sort-list>          ::=  { <sort> "," }+ .
<sort>               ::= <ident> .
<functions>          ::= "functions" <function>+ .
<function>           ::= <fun-ident> ":" <input-type> "->" <output-type> |
                         <operator> "_" ":" <sort> "->" <output-type> |
                         "_" <operator> "_" ":" <sort> "#" <sort> "->" <output-type> .
<fun-ident>          ::= <ident> .
<input-type>         ::= [ <product> ] .
<output-type>        ::= <product> .
<product>            ::= { <sort> "#" }+ .
<variables>          ::= "variables" <variable-list> .
<variable-list>      ::= ( <var-ident-list> ":" "->" <sort> )+ .
<var-ident-list>     ::= { <var-ident> "," }+ .
<var-ident>          ::= <ident> .
<equations>          ::= "equations" <cond-equation>+ .
<cond-equation>      ::= <tag> <equation-list> <implies> <equation> |
                         <tag> <equation> [ "when" <equation-list> ] .
<tag>                ::= "[" <ident> "]" .
<equation-list>      ::= { <equation> "," }+ .
<equation>           ::= <term> "=" <term> .
<term>               ::= [ <term> <operator> ] <primary> |
                         <operator> <primary> .
<primary>            ::= <fun-ident> ["(" <term-list> ")" ] |
                         <var-ident> | <tuple> |
                         "(" <term> ")" .
<term-list>          ::= { <term> "," }+ .
<tuple>              ::= "<" <term> "," <term-list> ">" .
```

2.2. Lexical syntax

Layout (1) or comment (2) may separate the following lexical notions of ASF: `<ident>` (3, 4), `<operator>` (5), and `<implies>` (6). Layout has no significance other than separating consecutive lexical tokens that would otherwise not be distinguished. Layout may never occur embedded in a lexical token. In cases of ambiguity, the longest lexical token is preferred. The lexical conventions of ASF are summarized below.

(1) Layout characters are space, horizontal tabulation, carriage return, line feed and form feed.

(2) Comments follow a layout character and begin with two hyphens and end with either an end of line (i.e., carriage return or line feed) or another pair of hyphens.

(3) Identifiers (i.e., `<ident>`) consist of a non-empty sequence of letters, digits or single quote characters, possibly with embedded hyphens. This is expressed by the following rules:

```
<id-char>      ::= <letter> | <digit> | "'" .
<ident>        ::= <id-char> [ ( <id-char> | "-" )* <id-char> ] .
```

For example, `x`, `max1`, `2-way`, `x''`, `double--hyphen`, `Very-Long-Identifier` and `6` are legal identifiers, but `-a`, `-` and `a-` are illegal.

(4) The following identifiers are reserved as keywords and cannot be used as an identifier:

```
begin      end         functions   parameters   to
bound      equations   imports     renamed      variables
by         exports     module      sorts        when
```

For technical reasons we also forbid the names `hidden` and `export` as `<parameter-ident>` (see section 2.6).

(5) Operators (i.e., `<operator>`) are denoted by either a sequence of one or more operator symbols or by an identifier surrounded by dots:

```
<op-symbol>      ::= "!" | "@" | "$" | "%" | "^" | "&" | "+" | "-" | "*" |
                     ";" | "?" | "~" | "/" | "|" | "\" .
<operator>       ::= <op-symbol>+ | "." <ident> "." .
```

The operators: `+`, `-`, `&&`, `.push.` and `!@%%?` are legal.

(6) The token `<implies>` consists of two or more consecutive `=` characters followed by either the character `>` or a new line:

```
<implies>        ::= "==" "="* ( ">" | "\n" ) .
```

2.3. Signatures, variables and equations

2.3.1. Signatures

Signatures are sets of declarations of sorts and functions over these sorts. Functions without arguments will also be called *constants*. See, for instance, [KLA83] or [EM85] for a description of signatures. The algebra of signatures and normalization of signature expressions are discussed in [BHK86]. The notion of signature used in ASF differs in three respects from the usual one:

- Functions, as defined in an ASF signature, may have various syntactical forms (see section 2.3.2).
- Functions may have tuples as output type (see section 2.4.1).
- Functions may be overloaded (see section 2.4.2).

A signature combined with a set of variables and a set of (positive conditional) equations forms a basic ASF module. Variables are typed with a sort in the signature.

In combination with a set of typed variables, a signature allows the construction of well-typed terms, i.e., terms obtained by type-wise correct composition of functions and variables. Due to the possibility of overloading, typing of terms is slightly more complicated than in the traditional case (see section 2.4).

Unconditional equations have the form:

$$[tag]\ t_l = t_r$$

where t_l and t_r are well-typed terms of the *same* type. Conditional equations can have two (equivalent) forms:

$$[tag]\ t_{l1} = t_{r1}, \ldots, t_{ln} = t_{rn}\ ==>\ t_l = t_r$$

or

$[tag]$ $t_l = t_r$ when $t_{l1} = t_{r1}, \ldots, t_{ln} = t_{rn}$

Variables in equations are implicitly universally quantified. Sound and complete rules of deduction for many-sorted conditional equations are given in [GM82].

2.3.2. Functions and operators

Depending on the way they are declared, functions are either

- ordinary prefix functions; or
- monadic prefix operators; or
- dyadic infix operators.

Declarations of prefix functions have the form:

```
<ident> ":" <input-type> "->" <output-type>
```

For instance,

```
f : S1 # S2 -> S3
```

defines a prefix function `f` with argument sorts `S1`, `S2` and output sort `S3`.

Prefix and infix operators may be used instead of, respectively, monadic and dyadic functions. The corresponding operator declarations have the following form:

```
    <operator> "_" ":" <sort>              "->" <output-type>
"_" <operator> "_" ":" <sort> "#" <sort> "->" <output-type>
```

The position of operands of operators is indicated by underline characters (_). For instance,

```
_ + _ : S1 # S2 -> S3
```

defines the infix operator + with argument sorts `S1`, `S2` and output sort `S3`, while

```
- _   : S1      -> S1
```

defines the monadic prefix operator - with both argument and output of sort `S1`. Infix and prefix operators are only a notational device and can always be replaced by ordinary functions. Dyadic operators are left-associative and have a lower priority than monadic ones.

2.3.3. The `if`-function

ASF provides a built-in conditional function `if`, which has the syntactic form of a `<primary>`. This function is polymorphic and cannot be defined in ASF itself. To a first approximation, `if` can be defined by the following signature schema:

```
sorts
   α, BOOL

functions
   if    : BOOL # α # α -> α
   true  :              -> BOOL
   false :              -> BOOL
variables
   a1, a2 : -> α
equations
[if.1]  if(true,  a1, a2) = a1
[if.2]  if(false, a1, a2) = a2
```

where `BOOL` corresponds to a predefined sort of Boolean values and α is a sort variable ranging over all sorts defined in a given specification including `BOOL`.

We prefer, however, to describe the meaning of `if` by showing how each conditional equation in which an `if` occurs can be replaced by *two* conditional equations from which that `if` has been eliminated. Assume that the conclusion or one of the conditions of a conditional equation e contains a (sub)term t_{if} = `if` (t_0, t_1, t_2). t_{if} can now be eliminated from e by replacing e by e' and e'', where

e' is obtained from e by replacing t_{if} by t_1 and by adding the condition t_0=`true`;

e'' is obtained from e by replacing t_{if} by t_2 and by adding the condition t_0=`false`.

Clearly, these steps can be repeated for all `ifs` occurring in e' and e''. A conditional equation containing n `ifs` can thus be replaced by 2^n conditional equations without `ifs`.

This method has the advantage that the data type of Boolean values can still be defined by the user provided that the specification contains the constants `true` and `false` of sort `BOOL`.

The definition of the `if`-function by means of a signature schema (the first method given above) is not equivalent to the definition based on `if`-elimination. The former leads to specifications containing an `if`-function for each sort, while the latter gives specifications containing no `if`-functions at all. We prefer the second definition since it amounts to a simple, local transformation of the specification.

2.4. Types

2.4.1. Tupled output types

In the signature tuples are allowed as output types, i.e., the function

```
f : S1 # S2 -> S3 # S4
```

has output type `S3 # S4`. Instances of this type are ordered pairs of values of sorts `S3` and `S4` respectively. In equations, tuples are written as a sequence of two or more terms enclosed by angle brackets < and > and separated by commas. The sorts of the components of such a tuple must be equal to the corresponding components of the applicable tupled output type in the signature. Tuples can be removed from the specification by introducing new sorts and construction functions for each tupled output type in the signature. The above tupled output type `S3 # S4` can be eliminated as follows:

- Introduce a new sort `S5` to act as a replacement for `S3 # S4`;
- Replace the definition of `f` by

  ```
  f : S1 # S2 -> S5
  ```

  and introduce the constructor

  ```
  make-S5 : S3 # S4 -> S5
  ```

- Replace all tuples of type `S3 # S4` by applications of `make-S5`, e.g. replace `<s3,s4>` by `make-s5(s3,s4)`.

Note, that no projection functions have to be defined, since the selection of elements from tuples can be accomplished by means of conditional equations. For instance, one can decompose the value of function `f` into components `x` and `y` (which are variables of sorts `S3` and `S4` respectively) as follows:

```
<x, y> = f(t1, t2) ==> ... equation using x and y ...
```

Tuples can only occur as output values. Hence, a tupled output value itself cannot be used directly as the argument of a function (but its components can, of course).

2.4.2. Overloading

Function names may be *overloaded*, i.e., the same function name may denote several functions with different types. For instance, after defining

```
f : S1 # S2 -> S3
f : S2      -> S2
```

each occurrence of the function name `f` in a term will have to be disambiguated by inferring its type from the types of the arguments to which it is applied. Since an overloaded name does not identify a specific function uniquely, we will in the sequel use *disambiguated* function names, which are obtained by postfixing function names with their type. Because input types of overloaded functions are required to be unique (see next section), it would be sufficient to use only the *input type* as a postfix.

2.4.3. Inside-out typing

Type assignment of a term is accomplished by inside-out (bottom-up) type assignment. This amounts to (1) determining the types of the constants and variables in a term; (2) propagating this type information (inside-out) to the enclosing terms until the type of the complete term has been determined. To guarantee the uniqueness of types during each stage of type assignment, the types of functions and variables must satisfy the following constraints:

- Overloaded functions (see section 2.4.2) should have unique input types (this forbids overloaded constants).
- Variables cannot be overloaded.
- The sets of constants and variables must be disjoint.

2.5. Exports, imports and parameters

A module may contain definitions of the following signatures:

- an *export* signature;
- zero or more *parameter* signatures;
- a *hidden* signature.

Each of these signatures may be *incomplete* in the sense that it may use sorts defined in one of the other signatures or in one of the signatures 'inherited' from imported modules (see below).

The export signature of a module is defined by an `exports` clause. The sorts and functions declared in it are visible outside the module. Hidden sorts and functions, on the other hand, are only visible inside the module in which they are declared. Hidden sorts are not permitted in the types of exported functions. Exported names are *inherited*, i.e., they are automatically exported by modules that (directly or indirectly) use the module from which the names were originally exported.

Import of a module in another module is the fundamental composition operation for modules. It is described by the `imports` clause. Importing module `B` in module `A` is equivalent to constructing a new module `A′` consisting of the union of the signatures and equations of `A` and `B`. Note, that the hidden names of module `B` are only visible inside `B` and can never clash with hidden or visible names of `A`. Technically, this can be achieved by renaming the hidden names of `B` to unique new names before modules `A` and `B` are combined. The declaration of the imported module must precede the declaration of the importing module. In this way cycles in the import graph are avoided.

In order to make modules more generally usable in different contexts, a form of parameterization is available in ASF. Parameterization is described by a `parameters` clause consisting of one or more formal parameters. Each parameter is a named (possibly incomplete) signature consisting of declarations of *formal* sorts and functions, which—at a later stage when the parameterized module is imported in another module—may be bound to *actual* ones (see section 2.7.2). Not all its parameters have to be bound before a module can be imported in another one. Such unbound parameters are *inherited* by the importing module and are indistinguishable from parameters that are specified in the importing module itself. Hence, the set of parameters of a module consists on the one hand of the parameters declared in the module itself (if any) and on the other hand of the parameters inherited from imported modules (if any). All these parameters should have different names. As parameter names are not susceptible to renaming, conflicts between (inherited) parameter names cannot be resolved within ASF itself but require editing of at least one of the names involved. To avoid direct or indirect self-binding, the declaration of a module must precede its use as an actual parameter.

We impose the following restrictions on parameters:

- The types of functions in parameters may not contain hidden sorts.
- Overloading of functions within a single parameter is not allowed (appendix I, examples 2.8 and 2.9).

The *visible* signature of a module is the complete set of names visible outside the module. It is the union of all its (locally declared as well as inherited) export and parameter signatures. As it may not use hidden names, the visible signature of a module must be complete.

Finally, the *internal* signature of a module is the union of its visible signature and its hidden signature. Like the visible signature, the internal signature must be complete. It consists of all names available inside the module.

2.6. The origin rule

When combining signatures of a module or modules of a specification, the problem arises how multiple declarations of the same name should be handled. Clearly, one wants to avoid random, i.e., unintended, name identifications. To this end we introduce the *origin rule.*

We associate with each (disambiguated) name a in the internal signature or variable section of a module an *origin* which is basically the textual position of a declaration of some (disambiguated) name n to which a 'owes its existence.' Thus, an origin is a tuple $[m,s,c,n]$, where

- m is the name of the module in which the declaration of n occurs;
- s is the section of the module in which the declaration of n occurs:

  $s = export$ for the export section,
  $s = hidden$ for the hidden and the variables section, and
  $s = p$ for a parameter section with name p.

- c is the subcategory to which n belongs:

  $c = sort$ for a sort name,
  $c = function$ for a function name, and
  $c = variable$ for a variable name.

- n is the (disambiguated) name introduced by the declaration.

Origins propagate in the following way:

(1) *Declaration*: At the moment a name a is declared, it obtains origin $[m,s,c,a]$, where m, s and c are determined from the context of the declaration. Hence, initially $n = a$.

(2) *Import*: Import of a name does not affect its origin.

(3) *Renaming*: A new name introduced by a renaming inherits the origin of the name it replaces. In general this leads to $n \neq a$. This applies to both explicit renaming (section 2.7.1) as well as implicit renaming (see below), but not to formal-to-actual renaming in a parameter binding.

(4) *Parameter binding*: The origin of an actual name does not change by binding a formal name to it (section 2.7.2). The origin of the formal name disappears along with the formal name itself.

ORIGIN RULE:

(1) Two visible sorts or functions are identical if they have both the same (disambiguated) name and the same origin. Visible sorts and functions having the same (disambiguated) name but different origin are forbidden.

(2) Two hidden sorts are identical if they have the same origin.

(3) Two variables are identical if they have the same origin and if the corresponding types (sorts) can be identified using (1) or (2).

(4) Two hidden functions are identical if they have the same origin and if the two corresponding types have equal structure and can be identified componentwise using (1) and (2).

The origin rule allows the multiple import of the same module (via different routes), but forbids clashes of identical (disambiguated) names originating from different modules or from different signatures within a module. Hidden names are *implicitly* renamed when modules are combined to avoid name clashes (section 4). The origin rule allows hidden names with the same origin to be identified, even if the names themselves are different.

Violations of the origin rule can always be eliminated by moving all declarations of conflicting names to a new module. This can be seen in the following example, where the declaration of sort `A` in both M1 and M2 causes a violation of the origin rule in M3, which imports both M1 and M2:

```
module M1
begin
   exports
      begin sorts A end
end M1

module M2
begin
   exports
      begin sorts A end
end M2

module M3
begin
   imports M1, M2
end M3
```

The origins associated with the declarations of sort A in modules M1 and M2 are [M1,*export*,*sort*,A] and [M2,*export*,*sort*,A] respectively. We can circumvent this violation of the origin rule by introducing a new module M0 in which sort A is declared, and by importing M0 in both M1 and M2:

```
module M0
begin
   exports
      begin sorts A end
end M0

module M1
begin
   imports M0
end M1

module M2
begin
   imports M0
end M2

module M3
begin
   imports M1, M2
end M3
```

This example shows that the origin rule enforces a certain modularization of ASF specifications.

2.7. Module expressions

Module expressions serve the purpose of renaming visible sorts and functions and of binding formal parameters to actual ones. After evaluation, the result of the module expression is imported in another module. The constituents of module expressions are described in the following subsections.

2.7.1. Renaming

Visible names of a module can be renamed by means of the `renamed by` construct, which specifies a renaming, i.e., a list of (old visible name, new visible name) pairs, to be applied to the module. The renaming should be consistent, i.e., lead to a correct new signature. In particular, the new signature should obey the origin rule. This implies that most name clashes due to renaming are forbidden, but that different names having the same origin may be renamed to the same name (appendix I, examples 1.16 and 1.17). The new name inherits the origin of the name it replaces.

Renamings do not allow the selective renaming of one of the instances of an overloaded function. In this case *all* instances are renamed simultaneously.

2.7.2. Parameter binding

Binding of parameters is achieved by the `bound by` construct, which specifies the name of a parameterized module, a parameter name, a list of bindings, i.e., (formal name, actual name)-pairs, and the name of an actual module. The effect of a parameter binding is that the parameter is replaced by the actual module as specified by the list of bindings. A parameter can thus be bound only once. The following rules apply:

- Actual names must belong to the visible signature of the actual module. In particular, parameter to parameter binding is allowed (appendix I, example 3.5).
- Formal names and actual names must be of the same kind, i.e., both should be either sort names or function names.
- The binding of formal functions should be consistent, i.e., the types of formal and actual functions should be equal modulo the binding of formal sorts.
- All sorts and functions of a parameter must be bound to a sort or function of the actual module.

The origin of actual names is not affected by binding.

3. EXAMPLE

3.1. A simple ASF specification

The following specification illustrates many of the features of ASF; it consists of four modules:

`Booleans`:
truth values with sort `BOOL`, constants `true` and `false`, and functions `and` and `or`. It uses a hidden function `not` to define `or`.

`Naturals`:
natural numbers with sort `NAT`, constant `0` and functions `succ` (successor) and `eq` (equality).

`Sequences`:
sequences of unspecified items (parameter sort `ITEM`) with constant `null` (the empty sequence), and functions `cons` (constructs a sequence given an item and a sequence) and `eq` (equality on sequences).

`Nstrings`:
sequences of natural numbers obtained by binding the parameter `Items` of `Sequences` to `Naturals`.

```
module Booleans
begin
   exports
      begin
         sorts BOOL
         functions
            true  :               -> BOOL
            false :               -> BOOL
            and   : BOOL # BOOL   -> BOOL
            or    : BOOL # BOOL   -> BOOL
      end
   functions
      not   : BOOL          -> BOOL

   variables
      x, y  : -> BOOL

   equations

   [B1] and(true, x)     = x
   [B2] and(false, x)    = false

   [B3] not(true)        = false
   [B4] not(false)       = true

   [B5] or(x, y)         = not(and(not(x), not(y)))

end Booleans

module Naturals
begin
   exports
      begin
         sorts NAT
         functions
            0    :               -> NAT
            succ : NAT           -> NAT
            eq   : NAT # NAT     -> BOOL
       end

   imports Booleans

   variables
      x, y : -> NAT

   equations

   [N1] eq(0, 0)                 = true
   [N2] eq(x, y)                 = eq(y, x)
   [N3] eq(succ(x), 0)           = false
   [N4] eq(succ(x), succ(y))     = eq(x, y)

end Naturals
```

```
module Sequences
begin
   parameters
      Items begin
               sorts   ITEM
               functions
                   eq : ITEM # ITEM -> BOOL
            end Items
   exports
      begin
         sorts SEQ
         functions
            null :               -> SEQ
            cons : ITEM # SEQ    -> SEQ
            eq   : SEQ # SEQ     -> BOOL
       end

   imports Booleans

   variables
      s, s1, s2  : -> SEQ
      i, i1, i2  : -> ITEM

   equations

   [S1] eq(null, null)                     = true
   [S2] eq(s1, s2)                         = eq(s2, s1)
   [S3] eq(cons(i, s), null)               = false
   [S4] eq(cons(i1, s1), cons(i2, s2))     = and(eq(i1, i2), eq(s1, s2))

end Sequences

module Nstrings
begin
   imports Sequences
                { renamed by
                        [ SEQ -> NSTRING,
                          null -> null-nstring]
                  Items bound by
                        [ ITEM -> NAT,
                          eq   -> eq]
                        to Naturals
                }
end Nstrings
```

Booleans is imported twice in Nstrings, once via Sequences and once via actual parameter Naturals. The origin rule permits this, and in the normal form of Nstrings the elements of Booleans are not duplicated.

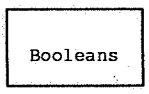
An example of the use of overloaded functions can be found in equation [S4] above. The first and the third occurrence of function eq have input type SEQ # SEQ, while the second occurrence of eq has input type ITEM # ITEM.

Function definitions in parameters (e.g. Items) only define a formal name for a particular function (e.g. eq). The equations for these functions can be found in the module to which the parameter is bound (e.g. equations [N1] - [N4], after the binding in module Nstrings).

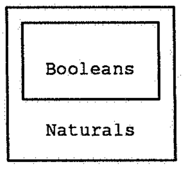
3.2. Structure diagrams

The overall modular structure of specifications can be illustrated by *structure diagrams*. These diagrams can be used effectively as a design aid.

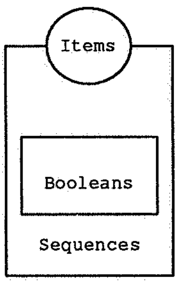
Each module is represented by a rectangular box in the structure diagram. The name of each module is shown at the bottom of its box. For example, module `Booleans` does not import any other module and is represented by the following structure diagram:



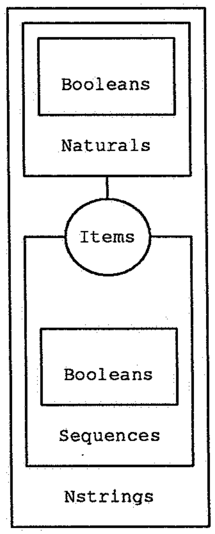
All modules imported by a module `M` are represented by structure diagrams inside the box representing `M`. For example, `Naturals` imports `Booleans` and is represented by:



For nested structure diagrams levels of detail may be suppressed to gain space. All parameters of a module are represented by ellipses carrying the name of the parameter. For example, `Sequences` (which has parameter `Items` and imports `Booleans`) is represented by:



The binding of a formal parameter is represented by a line joining the formal parameter and the actual module to which it is bound. For example, `NStrings` are defined by binding the parameter `Items` of `Sequences` to `Naturals`. The corresponding structure diagram is:



Inherited parameters are—not yet very satisfactorily—represented in structure diagrams by repeating the inherited parameter as a parameter of the module inheriting it (appendix I, example 2.1).

Structure diagrams can be generated automatically from the text of a given specification. This has been done in this report. Freek Wiedijk has designed a more concise graphical representation for the modular structure of specifications which is, unfortunately, not easily amenable to automation since it is based on finding a 'most planar' representation of a directed graph.

4. SEMANTICS OF ASF SPECIFICATIONS

4.1. Normalization of ASF modules

An ASF specification consists of a sequence of modules. A module can be evaluated in the context of a specification in which it occurs. This leads to a normal form, i.e., an ASF module from which all imports and as many parameters as possible have been eliminated. Normalization is an operation on the *text* of specifications.

We assign a semantics to each module in an ASF specification by assigning a semantics to its normal form (provided that it does not have unbound parameters), but not to the intermediate results that may occur during normalization.

4.1.1. Semantic domains

First, we introduce the domains MNAME, PNAME, SNAME, FNAME, VNAME and FTYPE of, respectively, *module names, parameter names, sort names, function names, variable names,* and *function types*. Although FTYPE could be defined using SNAME (since each function type consists of one or more sort names), we prefer to consider FTYPE as a primitive domain for reasons of simplicity. The domain SFV of sort names, (disambiguated) function names, and (disambiguated) variable names is defined as

$$\mathrm{SFV} = \mathrm{SNAME} \cup (\mathrm{FNAME} \times \mathrm{FTYPE}) \cup (\mathrm{VNAME} \times \mathrm{SNAME}).$$

The domain ORIGIN of *origins* is defined as

$$\mathrm{MNAME} \times (\{export, hidden\} \cup \mathrm{PNAME}) \times \{sort, function, variable\} \times \mathrm{SFV}.$$

The domain ORF of *origin functions* is defined as

$$\mathrm{ORF} = \mathrm{SFV} \to \mathrm{ORIGIN}.$$

The domain IR of *intermediate results* consists of modules *without imports, with possibly incomplete signatures,* and *possibly improper origin functions.* An intermediate result $ir \in \mathrm{IR}$ is a 6-tuple $(\Sigma, P, \Gamma, V, O, E)$, where

(1) Σ is a signature of exported sorts and functions.

(2) P is a list of pairs (p_i, Σ_i), where p_i is a parameter name and Σ_i is the corresponding parameter signature. The names p_i are different from each other. Both the export signature Σ and the parameter signatures Σ_i may use sorts declared in one of the others. Because *ir* does not contain imports, the union of these signatures is the *visible signature* of *ir* (section 2.5). The visible signature of *ir* need not be complete.

(3) Γ is a signature of hidden sorts and functions. Γ may use sorts defined in Σ or P. Because *ir* does not contain imports, the union of the visible signature of *ir* with Γ is the *internal signature* of *ir* (section 2.5). The internal signature of *ir* need not be complete.

(4) V is a set of typed variables over the internal signature of *ir*.

(5) $O \in \mathrm{ORF}$ is an origin function that associates an origin (as defined in section 2.6) with each sort and each disambiguated function name (section 2.4.2) defined in the internal signature, and with each variable in V. O may be improper in the sense that origins may contain names of formal parameters that no longer exist. Intermediate results always obey the origin rule.

(6) E is a set of conditional equations over a signature that may be larger than the internal signature of *ir*.

The domain $\mathrm{NF} \subset \mathrm{IR}$ of *normal forms* consists of all intermediate results *with complete signatures* and *proper origin functions*. Like intermediate results, *normal forms may have parameters*. More precisely, a normal form n is an intermediate result satisfying the following additional constraints:

(1) The visible signature of n is complete, i.e., all sorts used in function definitions in the visible signature are also declared in the visible signature.

(2) The internal signature of n is complete as well, i.e., all sorts used in function definitions in the internal signature and all functions used in equations are declared in the internal signature.

(3) All sorts and functions in the internal signature have proper origins. In particular, no origin may contain the name of a parameter that has been bound during normalization and thus no longer exists. (This simply means that all formal names of bound parameters have been eliminated.)

The domain ASF of ASF specifications consists of all strings derivable from the notion `<specification>` in the ASF grammar in section 2.1.

The domain RENAMING consists of non-empty lists of (*old-disambiguated-name*, *new-name*)-pairs:

$$\mathrm{RENAMING} = (\ (\mathrm{SNAME} \times \mathrm{SNAME}) \cup ((\mathrm{FNAME} \times \mathrm{FTYPE}) \times \mathrm{FNAME})\ )+.$$

4.1.2. Auxiliary functions

Depending on the order in which the elements of a module are evaluated, different results may be obtained. We evaluate the elements of a module in the same order in which they occur in the text. This strategy is not optimal from the viewpoint of normalization. It may lead to violations of the origin rule and other errors in some cases in which another evaluation order would produce a correct normal form. It has the advantage of being straightforward, however.

The *normalization function* $\mathrm{N} : \mathrm{MNAME} \times \mathrm{ASF} \rightarrow \mathrm{NF}$ computes the normal form of a module with a given name in the context of a given specification. To define N we need four auxiliary functions: *rename_visibles*, *rename_hiddens*, *combine*, and *bind*. All these functions are partial.

The types of the functions occurring in the following semi-formal description of the normalization procedure are to be understood as *dynamic constraints*. Each function is responsible for checking whether its own arguments and results have the proper type. If they do not, normalization fails. We have explicitly indicated most (but not necessarily all) other constraints that may cause normalization failures, so the normalization procedure can also be viewed as a static consistency checker for ASF modules.

4.1.2.1. *rename_visibles* : IR × RENAMING × ORF → IR

This function renames an intermediate result as indicated in a `renamed by` or `bound by` clause. The third argument of *rename_visibles* is the origin function of the actual module to be bound. It is undefined for ordinary renamings. Given an intermediate result $ir = (\Sigma, P, \Gamma, V, O, E)$ to be renamed, a renaming r, and a (possibly undefined) origin function O_{act}, $rename\_visibles(ir, r, O_{act})$ first applies a 'simultaneous' renaming to the visible signature of ir as prescribed by r. Σ, P and E may be affected by renamings of both sorts and functions (resulting in Σ', P' and E'), while Γ and V may be affected by the renaming of (visible) sorts only (resulting in Γ' and V'). The resulting origin function O' is determined as follows. Let $domain(O_{new}) = \varnothing$ and $O_{old} = O$. For each $(x, y) \in r$

(a) remove x from $domain(O_{old})$;

(b) define $O_{new}(y) = O(x)$ (ordinary renaming; O_{act} undefined; y inherits the origin of x), or

(b′) define $O_{new}(y) = O_{act}(y)$ (parameter binding).

Now take $O' = O_{old} \cup O_{new}$ (this fails if the result O' is not a function!) and define

$$rename\_visibles(ir, r, O_{act}) = (\Sigma', P', \Gamma', V', O', E').$$

Note that, if a new visible name clashes with an existing hidden name in Γ, the latter is *not* automatically renamed.

4.1.2.2. *rename_hiddens* : NF × IR → NF

This renaming does not correspond to an explicit `renamed by` or `bound by` clause, but is used implicitly by *combine* (next section). Given a normal form n to be renamed and an intermediate result ir, $rename\_hiddens(n, ir)$ applies a renaming to all hidden sorts and functions and all variables of n so that these names can no longer cause clashes when ir and n are merged by *combine*.

4.1.2.3. *combine* : IR × NF → NF

This function corresponds to an `import` clause. For an intermediate result ir and a normal form n to be imported in ir, *combine* is defined as

$$combine(ir, n) = ir \cup rename\_hiddens(n, ir),$$

where $\cup$ is componentwise union. As dictated by the origin rule (section 2.6), hidden names or variables of n that have the same origin as hidden names or variables of ir are identified, i.e., they are renamed to the names they have in ir. Only the hidden names of n are renamed, so the hidden names of ir can still clash with visible names of n (appendix I, example 1.12). This will be detected by the origin rule. The sets of parameter names of ir and n should be disjoint and the result of *combine* should be a normal form.

4.1.2.4. *bind* : NF × PNAME × RENAMING × NF → NF

This function corresponds to a `bound by` clause. Given a parameterized normal form $n_1 = (\Sigma_1, P_1, \Gamma_1, V_1, O_1, E_1)$, a parameter name $p \in P_1$, a renaming (binding) r, and an actual parameter in normal form $n_2 = (\Sigma_2, P_2, \Gamma_2, V_2, O_2, E_2)$, *bind* performs the following steps:

(a) add parameter signature Σ_p of n_1 to its export signature Σ_1, delete (p, Σ_p) from P_1, and let the result be ir (this is an intermediate result and not a normal form, because some names in ir may still have origins containing p, although this parameter does not exist in ir);

(b) let $ir' = rename\_visibles(ir, r, O_2)$;

(c) define $bind(n_1, p, r, n_2) = combine(ir', n_2)$.

For *bind* to succeed, each $(x, y) \in r$ should satisfy the following requirements:

- x is a sort name or a (function name, function type) pair defined in Σ_p;
- y is a name of a sort or function in the visible signature $\Sigma_2 \cup P_2$ of n_2;
- each name from Σ_p appears exactly once as old name in r.

4.1.3. The normalization function $\mathrm{N} : \mathrm{MNAME} \times \mathrm{ASF} \rightarrow \mathrm{NF}$

Given a module name $m \in \mathrm{MNAME}$ and a specification $S \in \mathrm{ASF}$, the normal form $\mathrm{N}(m,S)$ is computed with the aid of the auxiliary functions of the previous section as follows:

(1) Let M be the text of the module in S with name m (this step may fail if there is no such module).

(2) Let I be the intermediate result obtained from M by deleting its `imports` clause (if any) and by adding an origin function O (this step may fail if $I \notin \mathrm{IR}$).
$I = (\Sigma, P, \Gamma, V, O, E)$ with

- Σ an export signature consisting of the sorts and functions declared in the `exports` section of M;
- P a list of pairs (p_i, Σ_i), where p_i is the name of the i-th parameter of M and Σ_i is the corresponding parameter section of M;
- Γ a hidden signature consisting of the hidden sorts and functions declared in M;
- V the set of variables declared in M;
- O the origin function on $\Sigma \cup P \cup \Gamma \cup V$;
- E the set of equations in M obtained by eliminating all instances of `if` (section 2.3.3).

(3) Let the `imports` clause of M be

`imports` $E_1, \ldots, E_k$ $(k \geq 0)$

with module expressions

$E_i = m_i\ C_{i,1}, \ldots, C_{i,l}$ $(m_i \in \mathrm{MNAME}, l \geq 0, l$ depends on $i)$.

$C_{i,j}$ is either a `bound by` or a `renamed by` clause (the latter is only allowed for $j=1$ or $j=l$ and not both if $l \geq 2$).
For $i=1, \ldots, k$ evaluate E_i as follows:

(3a) Let $N_{i,0} = \mathrm{N}(m_i, S)$ be the normal form of m_i. To ensure termination the declaration of m_i must precede that of m in S.

(3b) For $j=1, \ldots, l$ evaluate $C_{i,j}$ as follows:

(3b1) If $C_{i,j}$ is a `bound by` clause

$p_{i,j}$ `bound by` $r_{i,j}$ `to` $m_{i,j}$, $(p_{i,j} \in \mathrm{PNAME}$, $r_{i,j}$ a renaming, $m_{i,j} \in \mathrm{MNAME})$

let

$N_{i,j} = \mathit{bind}(N_{i,j-1}, p_{i,j}, R_{i,j}, \mathrm{N}(m_{i,j}, S))$

be the normal form after binding $p_{i,j}$. The renaming $R_{i,j}$ is obtained from the list $r_{i,j}$ of (*old-name*, *new-name*)-pairs in $C_{i,j}$ by extending function names with their type. To ensure termination the declaration of $m_{i,j}$ must precede that of m_i in S.

(3b2) If $C_{i,j}$ is a `renamed by` clause

`renamed by` $r_{i,j}$, $(r_{i,j}$ a renaming)

let

$N_{i,j} = \mathit{rename\_visibles}(N_{i,j-1}, R_{i,j}, \mathit{undefined})$

be the normal form after renaming. As in the `bound by` case, the renaming $R_{i,j}$ is obtained from the list $r_{i,j}$ of (*old-name*, *new-name*)-pairs in $C_{i,j}$ by extending function names with their type. If *old-name* corresponds to an overloaded function, one pair is added to R for each instance of *old-name*.

(3c) Let $N_i = N_{i,l}$ be the normal form of E_i.

(4) The desired normal form is

$$\mathrm{N}(m,S) = \mathit{combine}(I, \mathit{combine}(\cdots \mathit{combine}(N_1, N_2) \cdots N_k)).$$

(By combining $N_1, \ldots, N_k$ first, we deviate slightly from the textual order of evaluation, which would correspond to

$combine\,(\cdots combine\,(combine\,(I\,,N_1),N_2)\cdots N_k)$.

The latter evaluation order only works if the type of *combine* is relaxed to $\mathrm{IR} \times \mathrm{NF} \rightarrow \mathrm{IR}$.)

The semantics of module m in the context of specification S is the initial algebra [EM85, MG85] of its normal form $\mathrm{N}(m\,,S\,)$, provided the latter has no void sorts (at least one closed term for each sort) and no unbound parameters.

4.2 An example of normalization

We show the result of normalizing module `Nstrings` in the context of the example specification given in section 3.1 (to which we will refer as *SPEC*). `Nstrings` involves both import and parameter binding.

Except for the origin information, which we add after each declaration, a normal form can be represented as an ordinary ASF module.

Upon import of `Booleans` in `Naturals`, variables `x` and `y` of sort `BOOL` are implicitly renamed to `X` and `Y` by *combine* (section 4.1.2.3) to avoid overloading with `x` and `y` of sort `NAT`. Function `eq` is overloaded in the normal form of `Nstring`, but this is allowed (see sections 2.4.2 and 2.4.3). Renaming of hidden function `not` of `Booleans` is not necessary in this example.

`Nstrings'` = N(`Nstrings`,*SPEC*) looks as follows:

```
module Nstrings'
begin
   exports
      begin
         sorts BOOL         [Booleans, export, sort, BOOL]
               NAT          [Naturals, export, sort, NAT]
               NSTRING      [Sequences, export, sort, SEQ]
         functions
            true          : -> BOOL
                            [Booleans, export, function, true : -> BOOL]
            false         : -> BOOL
                            [Booleans, export, function, false : -> BOOL]
            and           : BOOL # BOOL -> BOOL
                            [Booleans, export, function, and : BOOL # BOOL -> BOOL]
            or            : BOOL # BOOL -> BOOL
                            [Booleans, export, function, or : BOOL # BOOL -> BOOL]

            0             :  -> NAT
                            [Naturals, export, function, 0 : -> NAT]
            succ          : NAT -> NAT
                            [Naturals, export, function, succ : NAT -> NAT]
            eq            : NAT # NAT -> BOOL
                            [Naturals, export, function, eq : NAT # NAT -> BOOL]

            null-nstring  : -> NSTRING
                            [Sequences, export, function, null : -> SEQ]
            cons          : NAT # NSTRING -> NSTRING
                            [Sequences, export, function, cons : ITEM # SEQ -> SEQ]
            eq            : NSTRING # NSTRING -> BOOL
                            [Sequences, export, function, eq : SEQ # SEQ -> BOOL]
      end

   functions
      not : BOOL -> BOOL   [Booleans, hidden, function, not : BOOL -> BOOL]
```

```
  variables
     X  : -> BOOL        [Booleans, hidden, variable, x : -> BOOL]
     Y  : -> BOOL        [Booleans, hidden, variable, y : -> BOOL]
     x  : -> NAT         [Naturals, hidden, variable, x : -> NAT]
     y  : -> NAT         [Naturals, hidden, variable, y : -> NAT]
     s  : -> NSTRING     [Sequences, hidden, variable, s : -> SEQ]
     s1 : -> NSTRING     [Sequences, hidden, variable, s1 : -> SEQ]
     s2 : -> NSTRING     [Sequences, hidden, variable, s2 : -> SEQ]
     i  : -> NAT         [Sequences, hidden, variable, i : -> ITEM]
     i1 : -> NAT         [Sequences, hidden, variable, i1 : -> ITEM]
     i2 : -> NAT         [Sequences, hidden, variable, i2 : -> ITEM]

  equations

  [B1] and(true, X)                    = X
  [B2] and(false, X)                   = false
  [B3] not(true)                       = false
  [B4] not(false)                      = true
  [B5] or(X, Y)                        = not(and(not(X), not(Y)))

  [N1] eq(0, 0)                        = true
  [N2] eq(x, y)                        = eq(y, x)
  [N3] eq(succ(x), 0)                  = false
  [N4] eq(succ(x), succ(y))            = eq(x, y)

  [S1] eq(null-nstring, null-nstring)  = true
  [S2] eq(s1, s2)                      = eq(s2, s1)
  [S3] eq(cons(i, s), null-nstring)    = false
  [S4] eq(cons(i1, s1), cons(i2, s2))  = and(eq(i1, i2), eq(s1, s2))

end Nstrings'
```

5. COMMENTS ON ASF

5.1. Differences between ASF and the 'PICO-formalism'

In [BHK85] we have introduced a specification formalism that has become locally known as the 'PICO-formalism'. The differences between the PICO-formalism and ASF are:

- Some restrictions have been imposed on the use of overloaded functions.
- Polymorphic functions are no longer allowed. As a consequence, a polymorphic `if` function can no longer be defined but is now built-in.
- The origin rule has been reformulated: some name clashes are now allowed (e.g. due to renamings).
- Parameter sorts and functions may now be renamed (appendix I, example 2.12).
- Parameter to parameter binding is now allowed (appendix I, example 3.5).
- Identifiers may now also contain the single quote character (`'`).
- The lexical syntax for user-defined operators is more general.
- A `<tag>` was optional but is now required for each equation.
- Two new syntactic forms for conditional equations have been introduced.
- The parentheses surrounding the output type of functions with multiple output values are no longer necessary.
- There is no built-in notation for string constants.

5.2. Known defects and limitations of ASF

- All exported names are inherited by importing modules, i.e., they are also exported by the modules that directly or indirectly use the module from which the names were originally exported. This simple scheme has the undesirable property that the number of exported names never decreases and cannot be controlled. A more refined mechanism for hiding visible names is clearly desirable, but has not been included in ASF.
- It is not possible to rename or bind one of the instances of an overloaded function due to the lack of type information in renamings and bindings. For this reason, we do not allow overloaded functions in parameters.
- The inside-out typing scheme is too restrictive in some cases.
- In the text of a module the export signature precedes the import section. This has the advantage that the export signature has a fixed and prominent position in the text, but the disadvantage that the export and parameter signatures may contain sorts from imported modules appearing later in the text.
- It would be desirable to have a *general* facility for defining the syntax of operators and functions instead of the very limited possibilities offered by ASF. By means of such a facility specifications containing user-defined (conventional) notation for integer and string constants, sets, lists, etc. could be written. This would also eliminate the need for *ad hoc* conventions such as, for instance, the notation for string constants used in [BHK85]. The problem of introducing syntactic freedom in specifications is addressed in SDF [HK86a], [HK86b] and [HEN87].
- Some specifications could probably benefit from a notion of subsorts [GM86], i.e., an inclusion relationship between two or more sorts. This has to be modelled in ASF by explicit injection functions.
- ASF does not provide nested module definitions.
- All formal sorts and functions must be bound in a parameter binding; in most cases specification of a few of the bindings would be sufficient, since the other ones could be derived from the signatures of the formal and actual parameters.
- Parameter binding is restricted to the binding of a formal parameter to a single actual module. Interesting possibilities not covered by ASF are: (a) binding of a formal parameter to more than one actual module; (b) binding of a parameter to the module in which the parameterized module is imported.
- Binding a parameter of a parameterized module does not affect the origins of names that are not involved in the binding. As a consequence, names of different instances of a parameterized module are identified if they are equal, but this is clearly undesirable. The current version of the origin rule is too weak to handle these cases properly.
- The origin rule does not apply to parameter names (appendix I, example 2.4).
- Parameter names cannot be renamed.

5.3. Theoretical aspects of ASF

The prime reason for including hiding is that it is both practically and theoretically essential. See [BT82], [BT83] and [BT86] for theoretical results on the power of initial algebra specifications with and without hidden sorts and functions.

From [BK83] it follows that also for parameterized data types the features included in ASF are as strong as possible (i.e., every effective parameterized data type has an initial algebra specification with hidden sorts and functions).

The description of the semantics of specifications by means of a normalization procedure as given in section 4.1. is mathematically unsatisfactory. A better approach would be to translate ASF specifications into module algebra [BHK86, BHK87a,b]. In [BHK87b] it is shown that due to the limited form of hiding provided by ASF (all exported names are inherited by all importing modules) the origin rule has a true semantic meaning rather than only a formal syntactic one.

5.4. Related work

Some algebraic specification formalisms that have been documented in the literature are ASL [WIR83], PLUSS [GAU85], RAP-2 [HUS87], ACT-ONE [EM85], OBJ-2 [FGJM85], and LARCH [GH83]. ASF can be compared with an (algebraic) sublanguage of the wide-spectrum language COLD-S [JON84, JKR86]. The contributions of ASF for which some claim to novelty can perhaps be made are:

- The origin rule. It serves not only as a means to enforce modularization but also avoids numerous occurrences of renamed versions of the same hidden sort or function within one module. Nearly all context-sensitive constraints on ASF specifications can be expressed as violations of the origin rule.
- Structure diagrams, which display the structure of imports and parameters. These can be generated automatically from the text of an ASF specification.
- The precise description of the semantics of modular specifications by means of a syntactic normalization procedure, which replaces more complex semantical considerations. Of course, as pointed out in the previous section, this 'operational' semantics is not satisfactory in the long run due to lack of compositionality.
- The syntax and semantics of ASF have been described with as much precision as is needed to implement a parser and a static consistency checker. Freek Wiedijk has implemented such a checker in C. It should be noted that the interaction between imports, exports, parameters, parameter binding, hiding, overloading, renaming and the origin rule is so complex that most of the time we spent on the design of ASF went into analysing all problematic combinations of these features. Obviously, one would prefer a formalism that does not give rise to such complications, but we have not been able to design one that strikes a better balance between expressive power and simplicity.

ACKNOWLEDGEMENTS

We gratefully acknowledge the comments and suggestions made by Wiet Bouma, Niek van Diepen, Paul Hendriks, Volko de Jong, Monique Logger, Sjouke Mauw, Ard Verhoog, Pum Walters, and Freek Wiedijk. They all contributed, in one form or another, to the collective experience in writing algebraic specifications and, in particular, to our understanding of the underlying problems.

REFERENCES

[BHK85] J.A. Bergstra, J. Heering & P. Klint, "Algebraic definition of a simple programming language", Centre for Mathematics and Computer Science, Report CS-R8504, 1985.

[BHK86] J.A. Bergstra, J. Heering & P. Klint, "Module Algebra", Centre for Mathematics and Computer Science, Report CS-R8617, 1986.

[BHK87a] J.A. Bergstra, J. Heering & P. Klint, "Bi-interpretation models for the axioms of module algebra", Centre for Mathematics and Computer Science, to appear, 1987.

[BHK87b] J.A. Bergstra, J. Heering & P. Klint, "The origin rule, a structuring principle for designs in module algebra", Centre for Mathematics and Computer Science, to appear, 1987.

[BK83] J.A. Bergstra & J.W. Klop, "Initial algebra specifications for parametrized data types", *Elektronische Informationsverarbeitung und Kybernetik*, **19** (1983) 1/2, pp. 17-31.

[BT82] J.A. Bergstra & J.V. Tucker, "The completeness of the algebraic specification methods for computable data types", *Information and Control*, **54** (1982), 3, pp. 186-200.

[BT83] J.A. Bergstra & J.V. Tucker, "Initial and final algebra semantics for data type specifications: two characterization theorems", *SIAM Journal on Computing*, **12** (1983), 2, pp. 366-387.

[BT86] J.A. Bergstra & J.V. Tucker, "Algebraic specifications of computable and semi-computable data types", Report 2.86, Centre for Theoretical Computer Science, The University of Leeds, 1986; Report CS-R8619, Department of Computer Science, Centre for Mathematics and Computer Science, Amsterdam, 1986; to appear in *Theoretical Computer Science*.

[vD86] N.W.P. van Diepen, "A study in algebraic specification: a language with goto-statements", Centre for Mathematics and Computer Science, Report CS-R8627, 1986.

[EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specifications*, Vol. I, *Equations and Initial Semantics*, Springer-Verlag, 1985.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud & J. Meseguer, ''Principles of OBJ2'', *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp.52-66.

[GAU85] M.-C. Gaudel, ''Toward structured algebraic specifications'', in: *ESPRIT '85: Status Report of Continuing Work*, Part I, North-Holland, 1986, pp. 493-510.

[GH83] J.V. Guttag & J.J. Horning, ''Preliminary report on the Larch shared language'', Technical Report CSL-83-6, Xerox, Palo Alto, 1983.

[GM82] J.A. Goguen & J. Meseguer, ''Universal realization, persistent interconnection and implementation of abstract modules'', in *Proceedings 9th International Conference on Automata, Languages and Programming, Lecture Notes in Computer Science*, Vol. 134, pp. 265-281, Springer-Verlag, 1982.

[GM86] J.A. Goguen & J. Meseguer, ''Order-sorted algebra I: partial and overloaded operations, errors and inheritance'', Technical Report, SRI International, Computer Science Laboratory, to appear, 1986.

[HEN87] P.R.H. Hendriks, ''Type-checking Mini-ML: an algebraic specification with user-defined syntax'', Centre for Mathematics and Computer Science, to appear, 1987.

[HK86a] J. Heering & P. Klint, ''User-definable syntax for specification languages'' (preliminary version), in: *Generation of Interactive Programming Environments — GIPE, Intermediate Report*, Centre for Mathematics and Computer Science, Report CS-R8620, 1986.

[HK86b] J. Heering & P. Klint, ''A syntax definition formalism'', Centre for Mathematics and Computer Science, Report CS-R8633, 1986.

[HUS87] H. Hussmann, ''RAP-2 User Manual'', University of Passau, to appear, 1987.

[JON84] H.B.M. Jonkers, ''The single linguistic framework'', Technical Report Esprit Project FAST, 1984.

[JKR86] H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, ''A semantic framework for the COLD-family of languages'', Logic Group Preprint Series No. 9, Department of Philosophy, University of Utrecht, 1986.

[KLA83] H.A. Klaeren, *Algebraische Spezifikation: Eine Einführung*, Springer-Verlag, 1983.

[MAU87] S. Mauw, ''An algebraic specification of process algebra, including two examples'', FVI Report, University of Amsterdam, to appear, 1987.

[MG85] J. Meseguer & J.A. Goguen, ''Initiality, induction and computability'', in: M. Nivat & J.C. Reynolds, *Algebraic Methods in Semantics*, Cambridge University Press, 1985, pp. 460-541.

[WAL86] H.R. Walters, ''An annotated algebraic specification of the static semantics of POOL'', University of Amsterdam, Computer Science Department, Report FVI 86-20, 1986.

[WIR83] M. Wirsing, ''A Specification Language'', Habilitationschrift, Munich University, 1983.

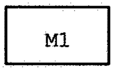
APPENDIX I. EXAMPLES OF ASF SPECIFICATIONS

In the following sections we give some examples of correct (''OK'') as well as incorrect (''KO'') ASF specifications. These examples illustrate various aspects of the formalism and may be used as (part of) a test set for an ASF static consistency checker.

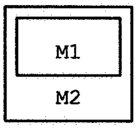
I.1. IMPORT/EXPORT/HIDING

I.1.1. [OK] Use of exported sorts/functions outside the module in which they are declared

```
module M1
begin
   exports begin sorts A end
end M1
```

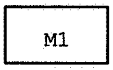


```
module M2
begin
   imports M1
   functions f : A -> A
end M2
```

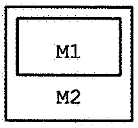


I.1.2. [KO] Use of hidden sorts/functions outside the module in which they are declared

```
module M1
begin
   sorts A
end M1
```

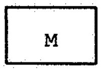


```
module M2
begin
   imports M1
   functions f : A -> A
   -- f has improper type, because
   -- sort A is not available in M2
end M2
```



I.1.3. [KO] Export of functions using hidden sorts

```
module M
begin
   exports begin functions f : A # A -> A end
   sorts A
end M
```

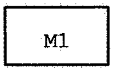


I.1.4. [OK] Hidden functions using exported sorts

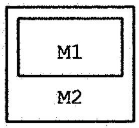
```
module M
begin
   exports begin sorts A end
   sorts B
   functions f : B # A -> A
end M
```

I.1.5. [OK] Inheritance of exported sorts/functions

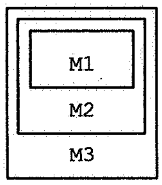
```
module M1
begin
   exports begin sorts A end
end M1
```



```
module M2
begin
   imports M1
   -- M2 inherits exported sort A from M1
end M2
```

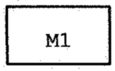


```
module M3
begin
   exports begin functions f : A # A -> A end
   imports M2
   -- M3 in turn inherits A from M2,
   -- so f is properly defined
end M3
```

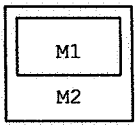


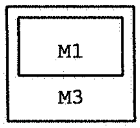
I.1.6. [OK] Multiple direct or indirect import of the same module

```
module M1
begin
   exports
      begin sorts A
            functions f : A -> A
      end
end M1
```

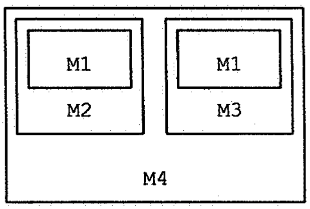


```
module M2
begin
   exports
      begin sorts B
            functions g : B -> A
      end
   imports M1
end M2
```





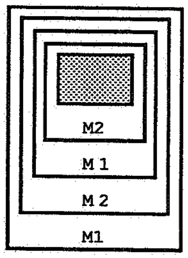
```
module M3
begin
   exports
      begin sorts C
            functions h : A # A -> C
      end
   imports M1
end M3
```

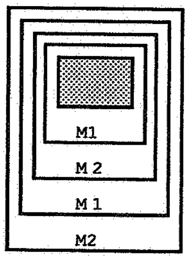


```
module M4
begin
   imports M2, M3
   -- M1 is imported twice in M4: via M2 and via M3;
   -- these two imports are identified by the origin rule
end M4
```

I.1.7. [KO] Direct or indirect self-import

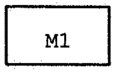
```
module M1
begin
   exports
      begin sorts A
            functions f : A -> B
      end
   imports M2
end M1

module M2
begin
   exports
      begin sorts B
            functions g : B -> A
      end
   imports M1
   -- Cycles in the import graph are not allowed
end M2
```

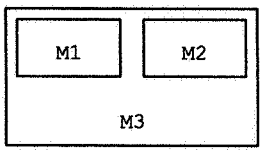




I.1.8. [KO] Overloading of constant due to import

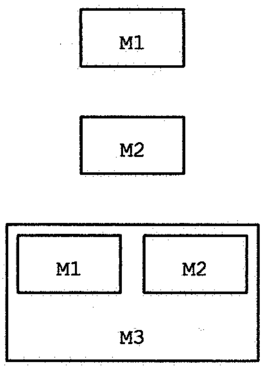
```
module M1
begin
   exports
      begin sorts A
            functions a :  -> A
      end
end M1

module M2
begin
   exports
      begin sorts B
            functions a :  -> B
      end
end M2

module M3
begin
   imports M1, M2
   -- Constant a becomes overloaded on import of M1 and M2 in M3
end M3
```





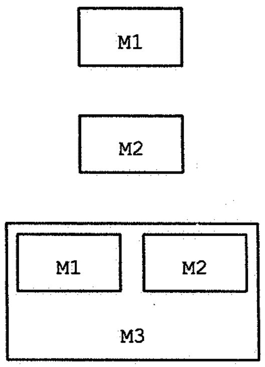
I.1.9. [OK] Overloading of non-constant function due to import

```
module M1
begin
   exports
      begin sorts A
            functions f : A -> A
      end
end M1

module M2
begin
   exports
      begin sorts B
            functions f : B -> B
      end
end M2

module M3
begin
   imports M1, M2
   -- Overloading of (non-constant) functions is allowed
end M3
```



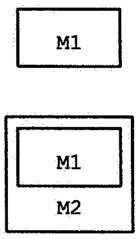
I.1.10. [KO] Clash of exported sorts/functions on import

```
module M1
begin
   exports
      begin sorts A
            functions f : A -> A
      end
end M1

module M2
begin
   exports
      begin sorts A
            functions f : A -> A
      end
end M2

module M3
begin
   imports M1, M2
   -- A and f : A -> A are imported both from M1 and M2,
   -- but this violates the origin rule
end M3
```



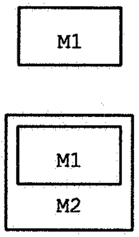
I.1.11. [OK] Hidden names of imported modules never cause name clashes

```
module M1
begin
   exports begin sorts A end
   sorts B
   functions f : A -> A
             g : B -> B
end M1

module M2
begin
   exports begin functions f : A -> A end
   imports M1
   sorts B
   functions g : B -> B
   -- B, g and f can be declared in M2 without causing
   -- clashes with identical but hidden names of M1
end M2
```



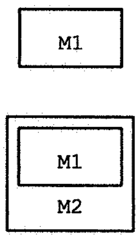
I.1.12. [KO] Exported function clashes with hidden function on import

```
module M1
begin
   exports
      begin sorts A
            functions f : A -> A
      end
end M1

module M2
begin
   imports M1
   functions f : A -> A
end M2
```



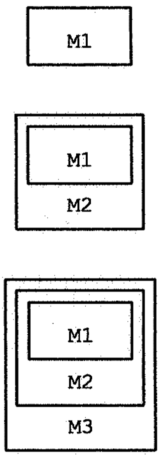
I.1.13. [OK] Renaming of exported sorts/functions

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
   end
end M1

module M2
begin
   imports M1 { renamed by [ A -> C,
                              B -> D,
                              f -> g ] }
end M2
```



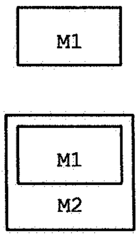
I.1.14. [OK] Renaming of exported sorts/functions

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
   end
end M1

module M2
begin
   imports M1
   -- A, B and f are inherited by M2
end M2

module M3
begin
   imports M2 { renamed by [ A -> C,
                             B -> D,
                             f -> g ] }
end M3
```



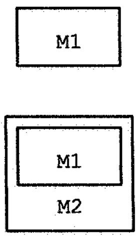
I.1.15. [KO] Renaming of hidden sorts/functions

```
module M1
begin
   exports begin sorts A end
   sorts B
   functions f : A -> B
end M1

module M2
begin
   imports M1 { renamed by [ B -> C,
                             f -> g ] }
end M2
```



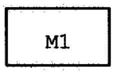
I.1.16. [KO] Name clash due to improper renaming

```
module M1
begin
   exports begin sorts A, B end
end M1

module M2
begin
   imports M1 { renamed by [ A -> C,
                             B -> C ] }
   -- C is imported twice in M2 with different origins
end M2
```

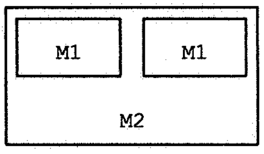


I.1.17. [OK] Non-injective renaming does not always cause a name clash

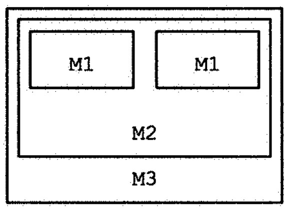
```
module M1
begin
   exports begin sorts A end
end M1
```



```
module M2
begin
   imports M1,
           M1 { renamed by [ A -> B ] }
end M2
```

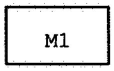


```
module M3
begin
   imports M2 { renamed by [ A -> C,
                             B -> C ] }
   -- C is imported twice in M4, but both
   -- instances have the same origin and are
   -- identified
end M3
```

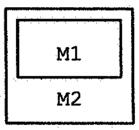


I.1.18. [OK] Permutative renaming

```
module M1
begin
   exports begin sorts A, B end
end M1
```



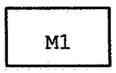
```
module M2
begin
   imports M1 { renamed by [ A -> B,
                             B -> A ] }
   -- Renamings are applied "simultaneously", so the
   -- order in which they are specified does not
   -- matter and the set of target names need not be
   -- disjoint from the set of source names
end M2
```

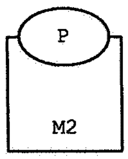


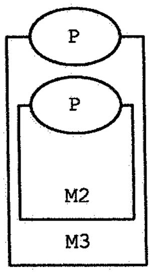
I.2. PARAMETERS

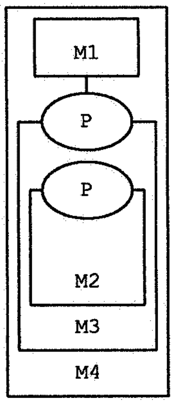
I.2.1. [OK] Inheritance of unbound parameters

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A # A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C # C -> D
        end P
end M2

module M3
begin
   imports M2
   -- M3 inherits parameter P from M2
end M3

module M4
begin
   imports M3 { P bound by [ C -> A,
                             D -> B,
                             g -> f ] to M1 }
   -- M4 does not inherit P, because P is bound
   -- on import of M3 in M4
end M4
```



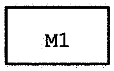


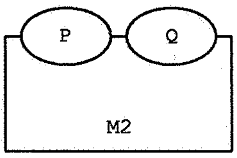


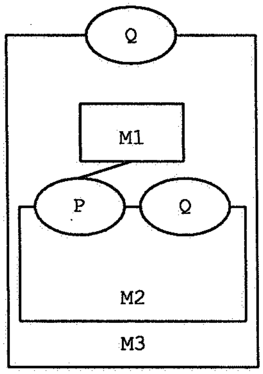


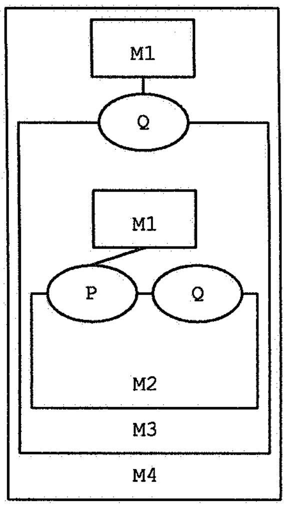
I.2.2. [OK] Inheritance of unbound parameters

```
module M1
begin
   exports
     begin sorts A, B
           functions a : -> A
                     f : A -> B
     end
end M1

module M2
begin
   parameters P begin sorts C
                      functions c : -> C
                end P,
              Q begin sorts D
                      functions g : C -> D
                end Q
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             c -> a ] to M1 }
   -- M3 inherits parameter Q from M2
end M3

module M4
begin
   imports M3 { Q bound by [ D -> B,
                             g -> f ] to M1 }
   -- Note that P and Q are both bound to M1;
   -- the origin rule allows this
end M4
```



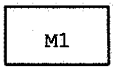




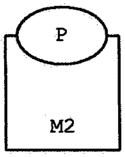


I.2.3. [OK] Inheritance of unbound parameters

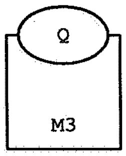
```
module M1
begin
   exports
     begin sorts A, B
           functions f : A -> B
     end
end M1
```



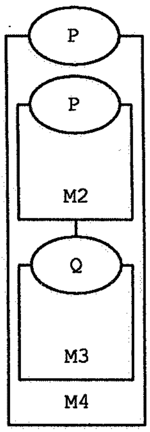
```
module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
        end P
   exports
      begin sorts E
            functions h : E -> E
      end
end M2
```



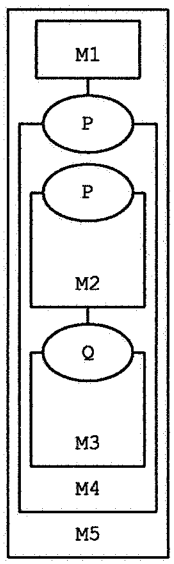
```
module M3
begin
   parameters
      Q begin sorts F
              functions i : F -> F
          end Q
end M3
```



```
module M4
begin
   imports M3 { Q bound by [ F -> E,
                             i -> h ] to M2 }
   -- M4 inherits parameter P from M2
end M4
```

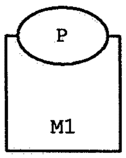


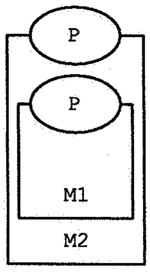
```
module M5
begin
   imports M4 { P bound by [ C -> A,
                             D -> B,
                             g -> f ] to M1 }
end M5
```

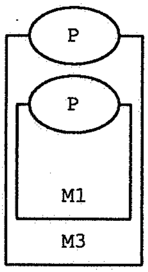


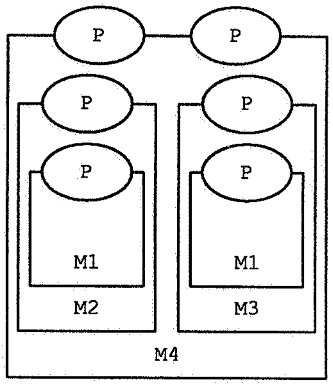
I.2.4. [KO] Multiple inheritance of the same parameter

```
module M1
begin
   parameters P begin sorts A end P
end M1

module M2
begin
   imports M1
   -- M2 inherits P from M1
end M2

module M3
begin
   imports M1
   -- M3 also inherits P from M1
end M3

module M4
begin
   imports M2, M3
   -- M4 inherits P both from M2 and M3;
   -- as the origin rule does not (yet)
   -- apply to parameter names, this is
   -- a name clash
end M4
```



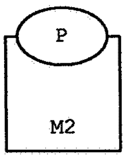


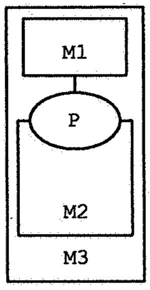




I.2.5. [OK] Inheritance of exported sorts/functions

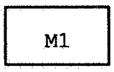
```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> A
      end
end M1

module M2
begin
   parameters
      P begin sorts C
              functions g : C -> C
        end P
end M2

module M3
begin
   exports begin functions h : B -> A end
   imports M2 { P bound by [ C -> A,
                             g -> f ] to M1 }
   -- M3 inherits the visible signature of M1,
   -- so h is correctly defined
end M3
```

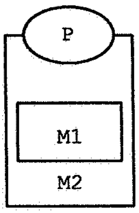




I.2.6. [OK] Function in parameter uses imported sorts

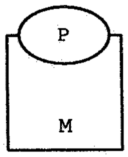
```
module M1
begin
   exports begin sorts A end
end M1

module M2
begin
   parameters
      P begin sorts B
              functions f : A # A -> B
        end P
   imports M1
end M2
```





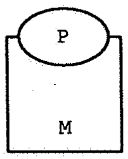
I.2.7. [KO] Function in parameter uses hidden sorts

```
module M
begin
   parameters
      P begin sorts A
              functions f : A -> B
        end P
   sorts B
end M
```



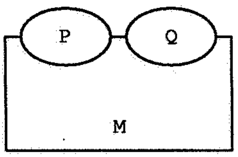
I.2.8. [KO] Overloading within parameter

```
module M
begin
   parameters
      P begin sorts A, B
              functions f : A -> B
                        f : A # A -> A
              -- Overloading of f within a single
              -- parameter is not allowed
        end P
end M
```



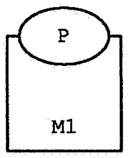
I.2.9. [OK] Parameters containing overloaded functions

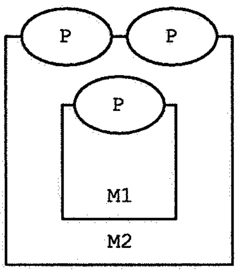
```
module M
begin
   parameters
      P begin sorts A
              functions f : A -> B
        end P,
      Q begin sorts B
              functions f : B -> A
        end Q
   -- Overloading of f in different parameters is OK
end M
```



I.2.10. [KO] Clash of parameter names on import

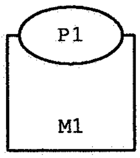
```
module M1
begin
   parameters P begin sorts A end P
end M1

module M2
begin
   parameters P begin sorts B end P
   imports M1
   -- M2 inherits parameter P from M1,
   -- but already has a parameter P itself
end M2
```



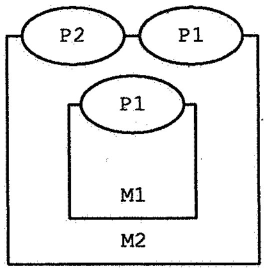


I.2.11. [KO] Clash of parameter sorts/functions on import

```
module M1
begin
   parameters
      P1 begin sorts A
                  functions f : A -> A
         end P1
end M1
```

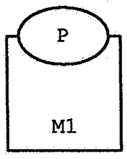


```
module M2
begin
   parameters
      P2 begin sorts A
                  functions f : A -> A
         end P2
   imports M1
end M2
```

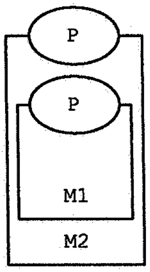


I.2.12. [OK] Renaming of parameter sorts/functions

```
module M1
begin
   parameters
      P begin sorts A, B
                functions f : A -> B
         end P
end M1
```

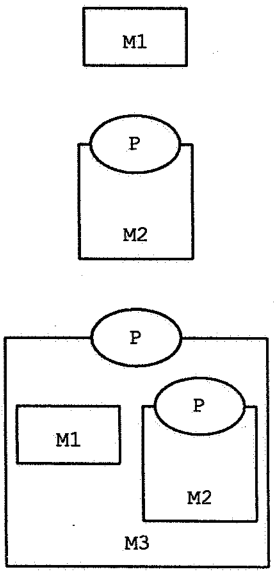


```
module M2
begin
   imports M1 { renamed by [ A -> C,
                             B -> D,
                             f -> g ] }

end M2
```



I.2.13. [KO] Name clash due to improper renaming

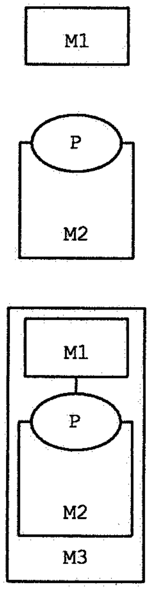
```
module M1
begin
   exports begin sorts A end
end M1

module M2
begin
   parameters P begin sorts B end P
end M2

module M3
begin
   imports M1 { renamed by [ A -> C ] },
           M2 { renamed by [ B -> C ] }
   -- C is imported twice in M3 with different
   -- origins
end M3
```



I.3. PARAMETER BINDING

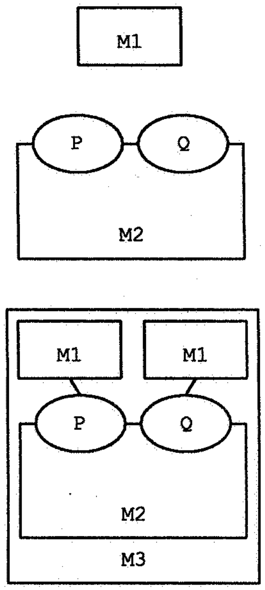
I.3.1. [OK] Binding of parameters

```
module M1
begin
   exports
      begin sorts A, B, C
            functions f : A # A -> B
                      g : A # B -> C
      end
end M1

module M2
begin
   parameters
      P begin sorts D, E
              functions h : D # D -> E
        end P
end M2

module M3
begin
   imports M2 { P bound by [ D -> A,
                             E -> B,
                             h -> f ] to M1 }
end M3
```



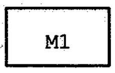
I.3.2. [OK] Multiple binding to the same module

```
module M1
begin
   exports
     begin sorts A, B
           functions a : -> A
                     f : B -> B
     end
end M1

module M2
begin
   parameters
      P begin sorts C
              functions c : -> C
        end P,
      Q begin sorts D
              functions g : D -> D
        end Q
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             c -> a ] to M1
                Q bound by [ D -> B,
                             g -> f ] to M1 }
   -- M1 enters M3 via two routes, but both
   -- instances are identified by the origin rule
end M3
```

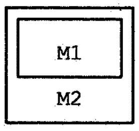


I.3.3. [OK] Binding of functions using imported sorts

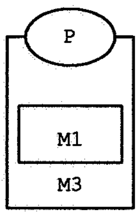
```
module M1
begin
   exports begin sorts A end
end M1
```



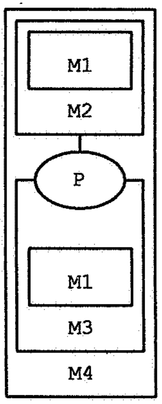
```
module M2
begin
   exports
      begin sorts B
            functions f : A # A -> B
      end
   imports M1
end M2
```



```
module M3
begin
   parameters
      P begin sorts C
                 functions g : A # A -> C
                 -- Sort A is imported from M1 and
                 -- cannot be bound
        end P
   imports M1
end M3
```

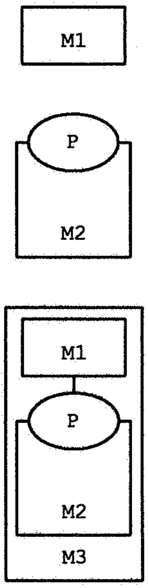


```
module M4
begin
   imports M3 { P bound by [ C -> B,
                              g -> f ] to M2 }
end M4
```



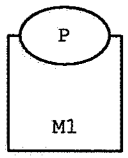
I.3.4. [KO] Binding to hidden sorts/functions

```
module M1
begin
   exports begin sorts A, B end
   functions f : A # A -> B
end M1

module M2
begin
   parameters
      P begin sorts C, D
                  functions g : C # C -> D
         end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             D -> B,
                             g -> f ] to M1 }
   -- f is not visible outside M1 and cannot
   -- be the target of a parameter binding
end M3
```

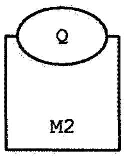


I.3.5. [OK] Parameter to parameter binding

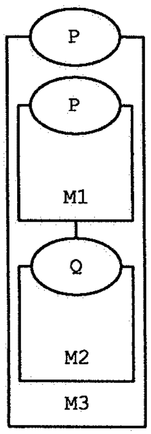
```
module M1
begin
   parameters
      P begin sorts A, B
              functions f : A -> A
                        g : B # B -> A
        end P
end M1
```



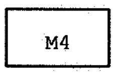
```
module M2
begin
   parameters
      Q begin sorts C
              functions h : C -> C
        end Q
end M2
```



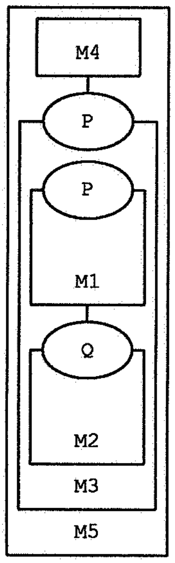
```
module M3
begin
   imports M2 { Q bound by [ C -> A,
                             h -> f ] to M1 }
   -- Parameter Q of M2 is bound to parameter P of M1,
   -- i.e. effectively becomes part of P
   -- P in turn is inherited by M3
end M3
```



```
module M4
begin
   exports
      begin sorts D, E
            functions i : D -> D
                      j : E # E -> D
      end
end M4
```

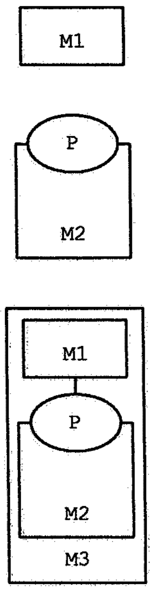


```
module M5
begin
   imports M3 { P bound by [ A -> D,
                             B -> E,
                             f -> i,
                             g -> j ] to M4 }
end M5
```



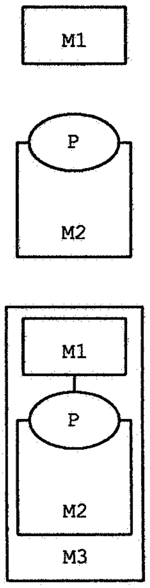
I.3.6. [KO] Incomplete parameter binding

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
                        h : D -> C
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             D -> B,
                             g -> f ] to M1 }
   -- h should have been bound too
end M3
```



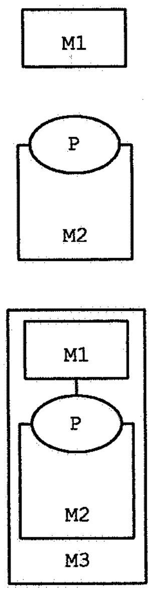
I.3.7. [KO] Inconsistent parameter binding

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A # A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             D -> B,
                             g -> f ] to M1 }
   -- The types of g and f are incompatible
end M3
```



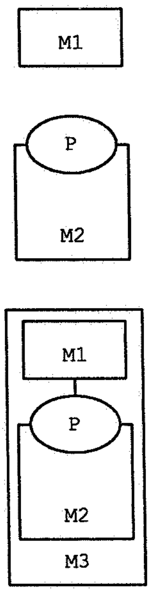
I.3.8. [KO] Inconsistent parameter binding

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> B,
                              D -> A,
                              g -> f ] to M1 }
   -- g -> f implies C -> A and D -> B
end M3
```



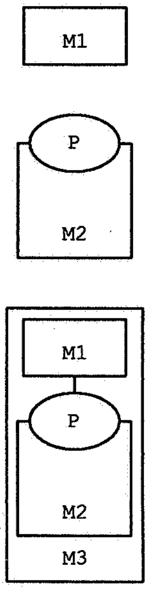
I.3.9. [OK] Binding to overloaded function

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
                      f : B -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
                        h : D -> D
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                              D -> B,
                              g -> f,
                              h -> f ] to M1 }
end M3
```



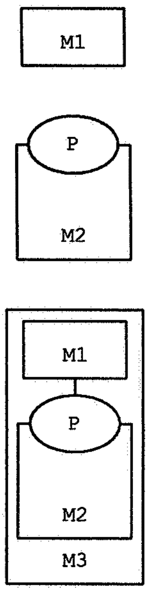
I.3.10. [KO] Multiple (i.e. non-injective) binding to same target sort/function

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions g : C -> D
                        h : C -> D
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             D -> B,
                             g -> f,
                             h -> f ] to M1 }
   -- g and h are both bound to f
end M3
```



I.3.11. [KO] Multiple binding of sort/function

```
module M1
begin
   exports
      begin sorts A, B
            functions f : A -> B
                      g : A -> B
      end
end M1

module M2
begin
   parameters
      P begin sorts C, D
              functions h : C -> D
        end P
end M2

module M3
begin
   imports M2 { P bound by [ C -> A,
                             D -> B,
                             h -> f,
                             h -> g ] to M1 }
   -- h is bound to both _ and g
end M3
```


M1
P
M2
M1
P
M2
M3